Spell Checker

Jonas P. de Oliveira Mat: 20180039036

Nathan C. de M. Gomes. Mat: 20180040977

Thomas R. de Araújo Mat: 20180039733

Resumo

Este trabalho tem como objetivo desenvolver uma ferramenta conhecida como spellchecker (Verificador Ortográfico) utilizando tabelas hash. O spell checker é um programa que verifica palavras de um texto e avalia com um dicionário, se uma determinada palavra do texto é encontrada no dicionário, presume-se que ela está escrita corretamente senão, considera-se que ela esteja escrita de forma incorreta ou que o dicionário não contém determinada palavra.

1 .Introdução

No decorrer do trabalho desenvolvemos diversos tipos de algoritmos para compararmos qual é o mais eficiente, dentre eles os métodos: *Linear probing, quadratic probing e chaining*.

2. Função de Hash

Uma função de hash tem como objetivo transformar o conteúdo de entrada em um valor inteiro onde retornará um indíce de sua posição na tabela hash.

A função de hash, é utilizada para melhorar desempenhos em relação ao tempo de inserção e pesquisa, onde é fundamental em problemas com muitos dados.

A função de hash utilizada no trabalho é chamada “djb2” onde consiste na multiplicação de um número primo muito grande (recomenda-se usar 5381) e é adicionado 33 ao valor de cada letra do código ASCII.

Como mostra na figura:: .

```c
//Funcao de hash djb2
unsigned int hash(char *word, int M){
    unsigned int value = 5381;//

    int i=0;
    for (i = 0; i < M; i++){
        value = value * 33  + word[i];
    }
    return value;
}
```

Quanto melhor a função de hash, melhor o espalhamento entre os buckets, porém, por mais que a função de hash seja boa, sempre irá existir mais de duas colisões em cada bucket.

3. Tratamento de colisão

Colisão é quando ocorre quando o cálculo do hash é igual, ou seja dois dados ocupando a mesma posição, para isso existe vários métodos para resolver esse impasse. Os métodos que foram feitos neste trabalho foram: Chaining (encadeamento), Linear Probing e Quadratic probing.

3.1 Encadeamento (Chaining)

Dentre as formas usadas para implementação do spellchecker usamos o método

de encadeamento (chaining) para solucionar nosso problema de colisão. Com isso cada elemento da tabela de hash é uma lista encadeada, alocada dinamicamente de acordo com a quantidade de elementos inseridos na mesma posição. Na pesquisa calculamos o hash do determinado dado, verificamos no decorrer da lista encadeada se o dado desejado está contido ou não. Observe o funcionamento deste método de colisão:

Inserção:

```
    //verifica se a posicao esta vazia
    if(table->nodes[indice] == NULL){
        table->nodes[indice] = new_node;
    }else{//Se nao estiver vazia, encadeia
        struct Node *aux = (struct Node*) malloc(sizeof(struct Node)); //criando uma auxiliar
        aux->word = (char*) malloc(sizeof(char)*(strlen(string)+1));

        strcpy(aux->word, string);
        aux->next = table->nodes[indice]; //colocando a palavra no comeco
        table->nodes[indice] = aux; //encadeando
    }
}
```

Quando vamos inserir um elemento (nó) na nossa tabela o custo é constante, pois só é necessário dar um “prepend” (colocar o elemento na frente da lista), o mesmo procedimento é feito quando ocorre uma colisão.

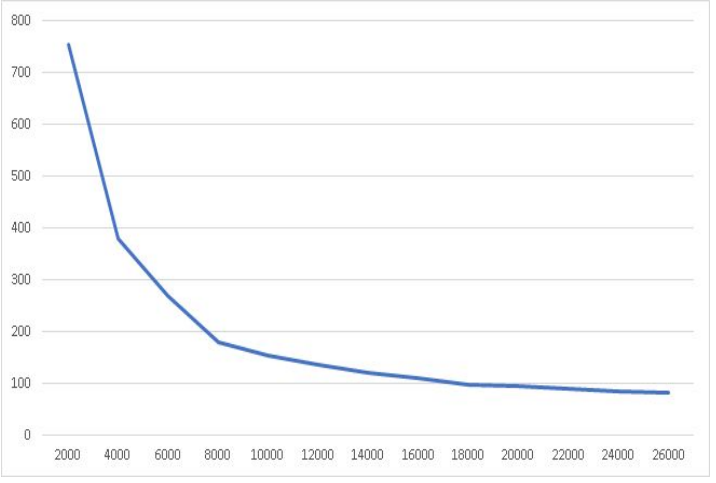
Pesquisa:

```
//Funcao de busca de palavra no dicionario
int chaining_word_search(char* word, struct HashTable *hash_table){
    unsigned int a = hash(word,strlen( void *malloc(size_t _Size) /Hash da palavra
    struct Node *aux = (struct Node*) malloc(sizeof(struct Node));
    aux->word = (char*) malloc(sizeof(char)*(strlen(word)+1)); //criando auxiliar
    aux = hash_table->nodes[a];

    if(aux == NULL){
        return 1;
    }else{
        while(aux != NULL){
            if(strcmp(aux->word, word) == 0){
                return 0;
            }else{ //se nao achar a palavra, vai para a proxima encadeada
                aux = aux->next;
            }
        }
        return 1;
    }
}
```

Por outro lado, o custo de pesquisa varia com a função de hash utilizada, se ele distribui bem ou não os itens, e com a quantidade de buckets.

Variação do tempo de pesquisa com relação aos buckets.



No eixo x temos o número de buckets e no eixo y o tempo em ms.

Podemos ver que começa a se estabilizar em 22000 buckets e 100 ms.

3.2 Linear probing

O Linear probing é uma técnica de tratamento de colisões onde os dados armazenados em um array, caso haja uma colisão no hash desse índice, a posição será incrementada em 1 unidade até ser encontrado o bucket vazio. A pesquisa será calcular o valor de hash e se aquela posição for uma posição ocupada e não for o dado desejado, terá que ser feito o incremento e comparar até achar aquele item desejado, caso encontre um bucket vazio no percorrimento em busca do dado desejado, significa que o dado não existe na tabela.

Inserção:

```
    //verifica se a posicao esta vazia
    if(table->nodes[indice] == NULL){
        table->nodes[indice] = new_node;
    }else{//Se nao estiver vazia procura a proxima vazia
        while(table->nodes[indice] != NULL){
            ++indice;
            indice = indice%table->buckets;
        }
        table->nodes[indice] = new_node;
    }
}
fclose(stream);
```

Busca:

```
int linear_word_search(char* word, struct HashTable *hash_table){
    unsigned int a = hash(word,strlen(word))%hash_table->buckets;//Hash da palavra

    if(hash_table->nodes[a] == NULL){
        return 1;
    }else{
        while(hash_table->nodes[a] != NULL){
            if(strcmp(hash_table->nodes[a]->word,word) == 0){
                return 0;
            }else{
                ++a;//proximo indice da palavra
                a = a%hash_table->buckets;
            }
        }
        return 1;
```

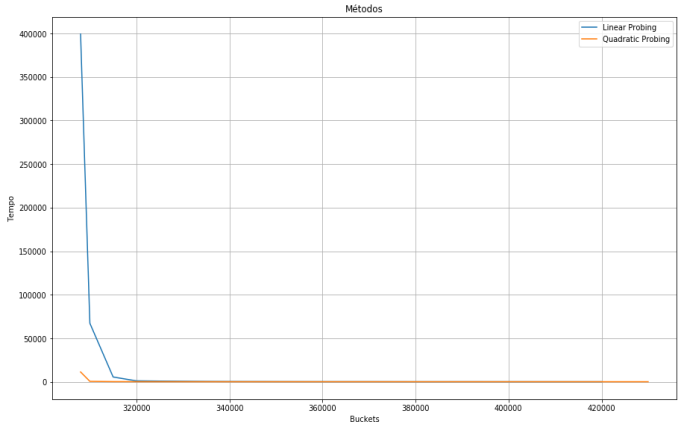
Quanto ao custo da inserção e da pesquisa não dá para atribuir um valor, pois seria constante. Caso não tenha nenhuma colisão (mas como é difícil não ocorrer colisões) o custo passa a ser uma probabilidade de alguém está ocupando ou não o bucket correspondente daquele item e os seus subsequentes.

3.3 Quadratic probing

O quadratic probing é semelhante a técnica do Linear probing, a alteração é feita no incremento no momento de colisão, passando a ser o incremento em 1 unidade e calculando o quadrado de seu número.

É calculado o hash novamente e verificando na posição se está contido ou não, caso não esteja verifica na próxima posição que é o quadrado do hash, caso não seja verificado e encontrar um bucket vazio na incrementação do hash, significa que o dado buscado, não está contido.

Segue abaixo a comparação do Linear probing e Quadratic probing.



Podemos ver que o Quadratic probing estabiliza mais rápido

4.Conclusão:

Para concluirmos, verificamos o tempo gasto de cada técnica usada, segue abaixo todos os melhores resultados obtidos dos métodos apresentados:

O dicionário utilizado disponibilizado pelo professor, contendo **307855 palavras** e o texto

comparado com o dicionário contém **83849 palavras**.

Assim concluímos que o melhor resultado em questão de tempo foi o método Quadratic probing com 29 ms e o Linear Probing com 30 ms. Mesmo não sendo uma diferença muito grande comparado ao Linear probing, o Quadratic probing estabiliza mais rápido. Enquanto o pior caso foi o chaining, com o melhor resultado de pesquisa com 46 ms, porém, o tempo gasto para carregar o dicionário foi o melhor entre os três.

5. Referências:

Tabelas Hash: Slides disponibilizados pelo professor.

Função de hash djb2. Diponível em <http://www.cse.yorku.ca/~oz/hash.html>

Arquivo usando para os testes. Diponível em <http://www.planalto.gov.br/ccivil_03/constituicao/constituicao.htm>